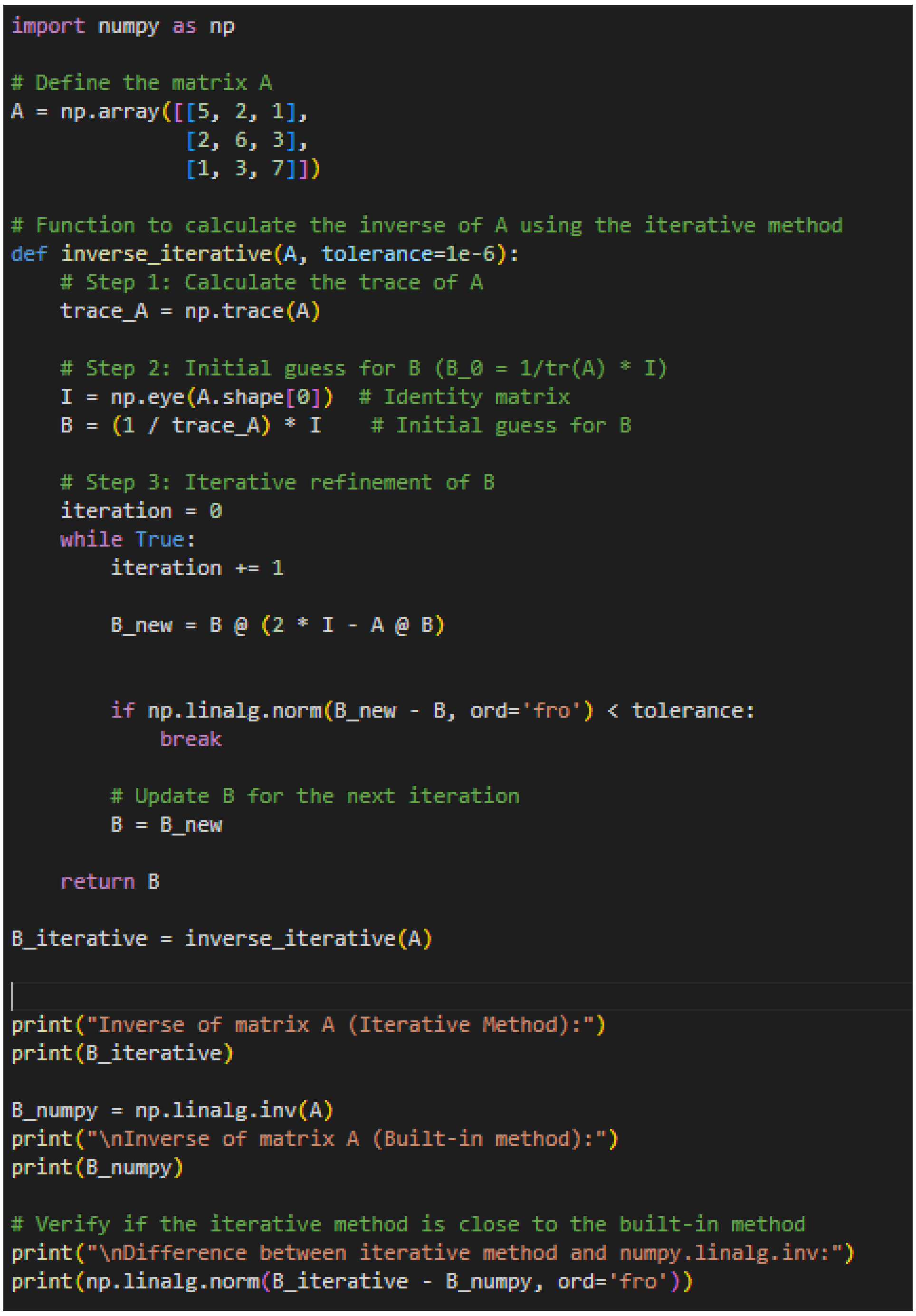
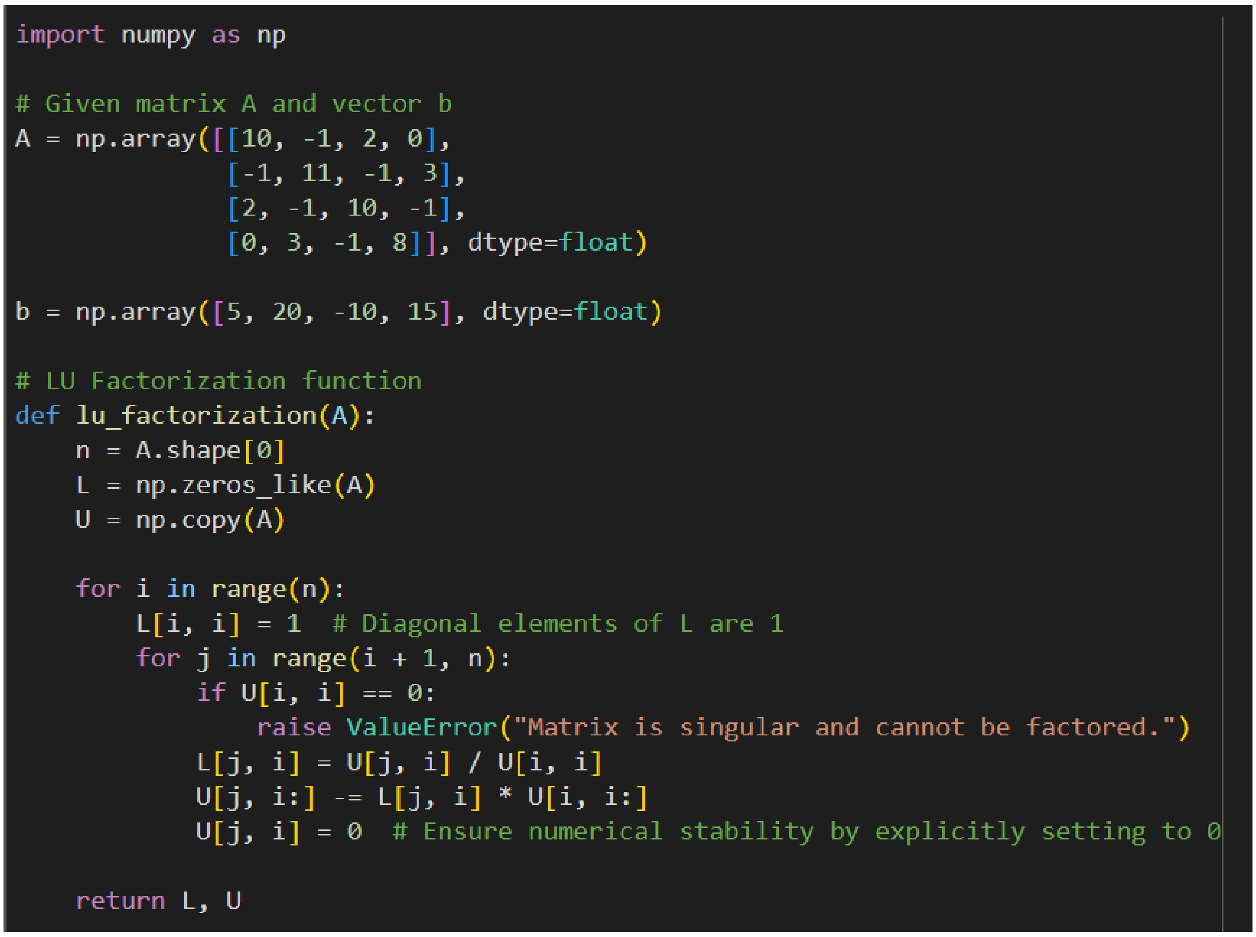
Assignment 3

Task #1

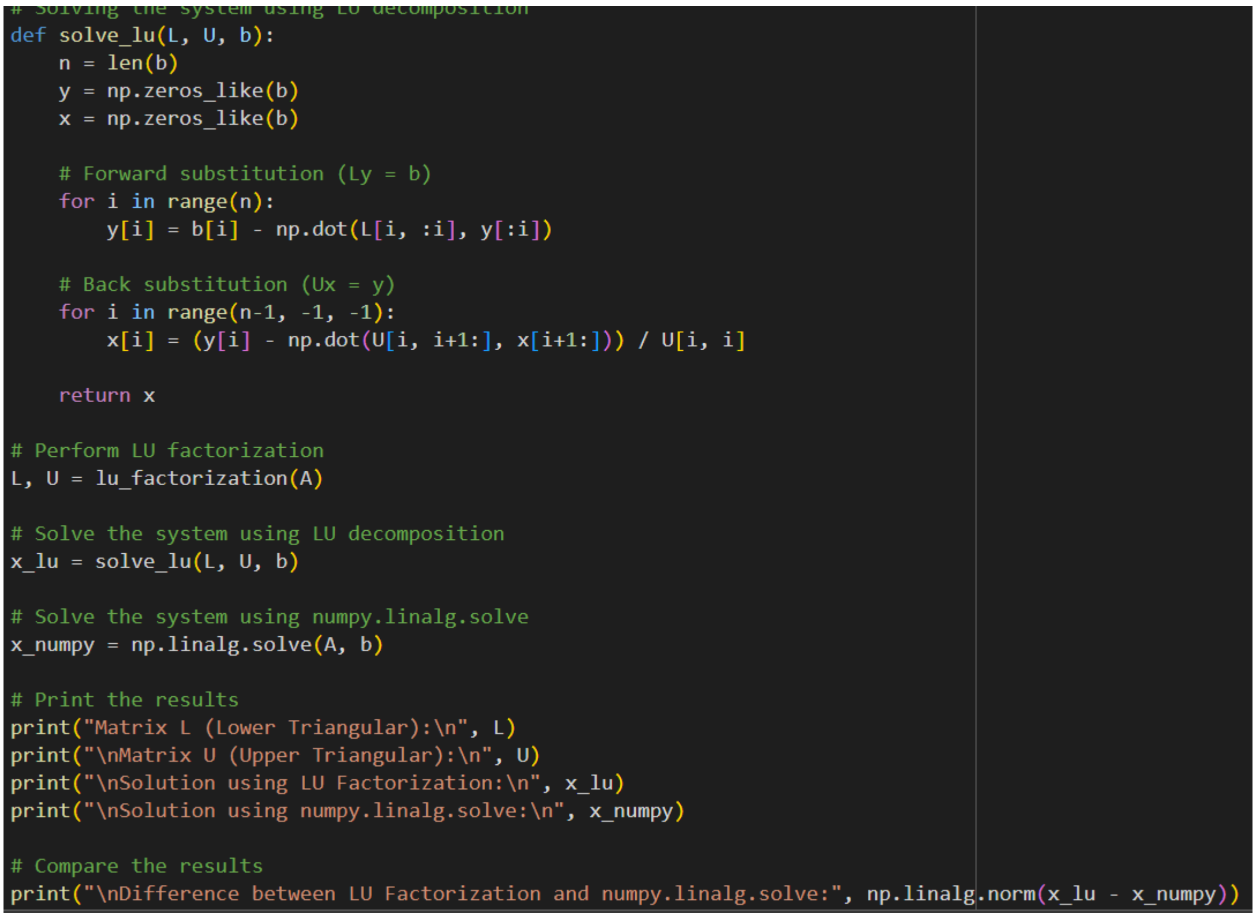


```python
import numpy as np

# Define the matrix A
A = np.array([[5, 2, 1],
              [2, 6, 3],
              [1, 3, 7]])

# Function to calculate the inverse of A using the iterative method
def inverse_iterative(A, tolerance=1e-6):
    # Step 1: Calculate the trace of A
    trace_A = np.trace(A)

    # Step 2: Initial guess for B (B_0 = 1/tr(A) * I)
    I = np.eye(A.shape[0])  # Identity matrix
    B = (1 / trace_A) * I    # Initial guess for B

    # Step 3: Iterative refinement of B
    iteration = 0
    while True:
        iteration += 1

        B_new = B @ (2 * I - A @ B)


        if np.linalg.norm(B_new - B, ord='fro') < tolerance:
            break

        # Update B for the next iteration
        B = B_new

    return B

B_iterative = inverse_iterative(A)


print("Inverse of matrix A (Iterative Method):")
print(B_iterative)

B_numpy = np.linalg.inv(A)
print("\nInverse of matrix A (Built-in method):")
print(B_numpy)

# Verify if the iterative method is close to the built-in method
print("\nDifference between iterative method and numpy.linalg.inv:")
print(np.linalg.norm(B_iterative - B_numpy, ord='fro'))
```

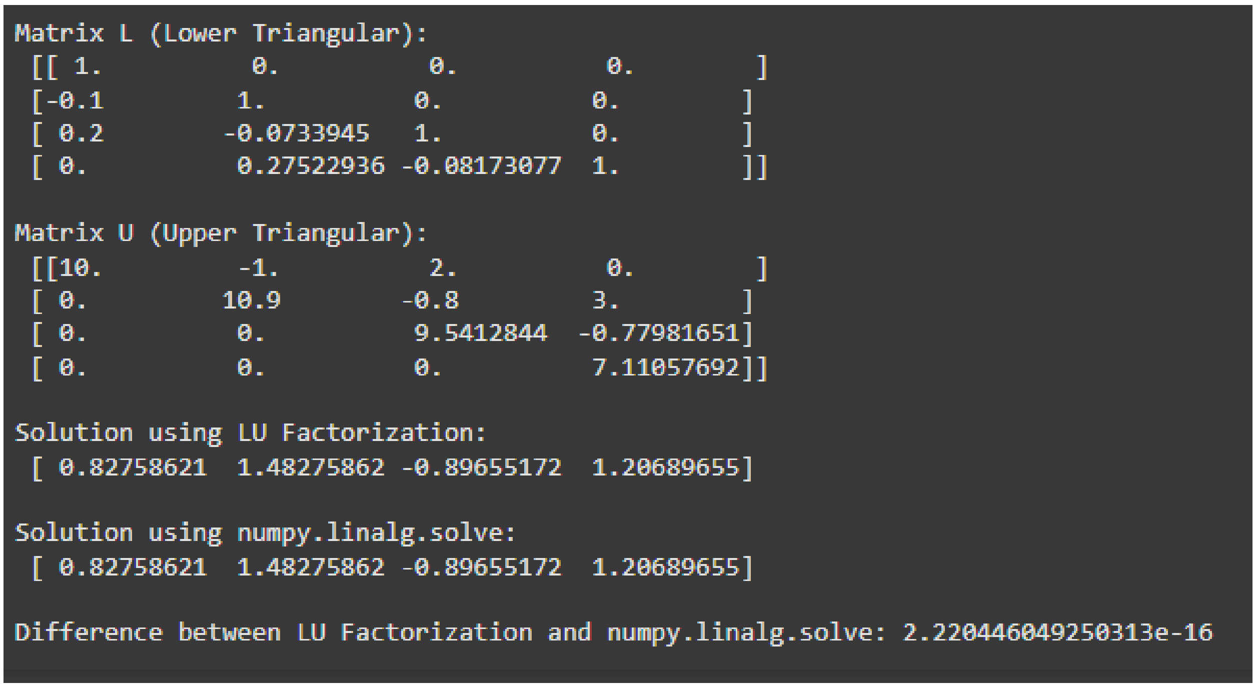
Task #2



```python
import numpy as np

# Given matrix A and vector b
A = np.array([[10, -1, 2, 0],
              [-1, 11, -1, 3],
              [2, -1, 10, -1],
              [0, 3, -1, 8]], dtype=float)

b = np.array([5, 20, -10, 15], dtype=float)

# LU Factorization function
def lu_factorization(A):
    n = A.shape[0]
    L = np.zeros_like(A)
    U = np.copy(A)

    for i in range(n):
        L[i, i] = 1  # Diagonal elements of L are 1
        for j in range(i + 1, n):
            if U[i, i] == 0:
                raise ValueError("Matrix is singular and cannot be factored.")
            L[j, i] = U[j, i] / U[i, i]
            U[j, i:] -= L[j, i] * U[i, i:]
            U[j, i] = 0  # Ensure numerical stability by explicitly setting to 0

    return L, U
```

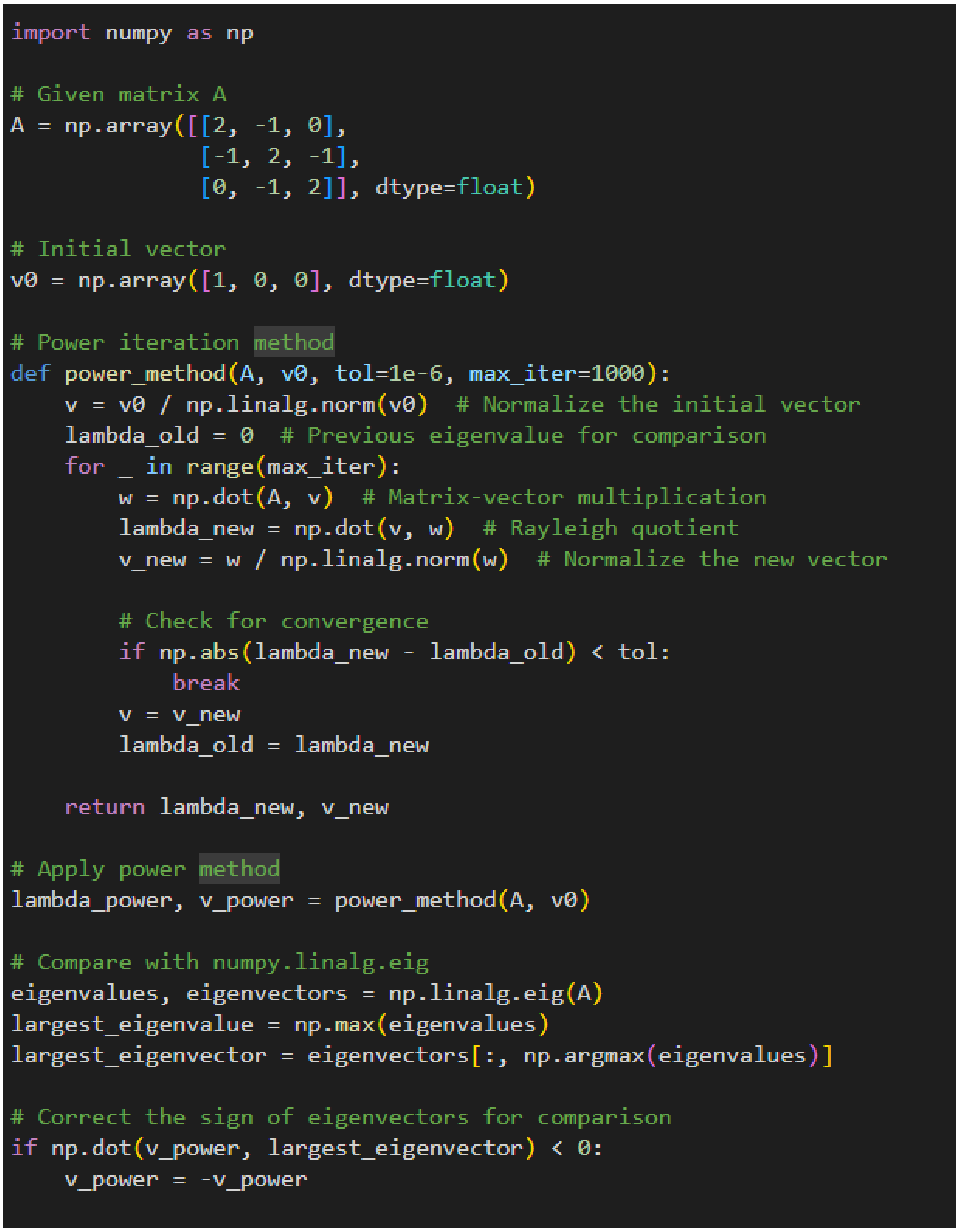


```python
# Solving the system using LU decomposition
def solve_lu(L, U, b):
    n = len(b)
    y = np.zeros_like(b)
    x = np.zeros_like(b)

    # Forward substitution (Ly = b)
    for i in range(n):
        y[i] = b[i] - np.dot(L[i, :i], y[:i])

    # Back substitution (Ux = y)
    for i in range(n-1, -1, -1):
        x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]

    return x

# Perform LU factorization
L, U = lu_factorization(A)

# Solve the system using LU decomposition
x_lu = solve_lu(L, U, b)

# Solve the system using numpy.linalg.solve
x_numpy = np.linalg.solve(A, b)

# Print the results
print("Matrix L (Lower Triangular):\n", L)
print("\nMatrix U (Upper Triangular):\n", U)
print("\nSolution using LU Factorization:\n", x_lu)
print("\nSolution using numpy.linalg.solve:\n", x_numpy)

# Compare the results
print("\nDifference between LU Factorization and numpy.linalg.solve:", np.linalg.norm(x_lu - x_numpy))
```

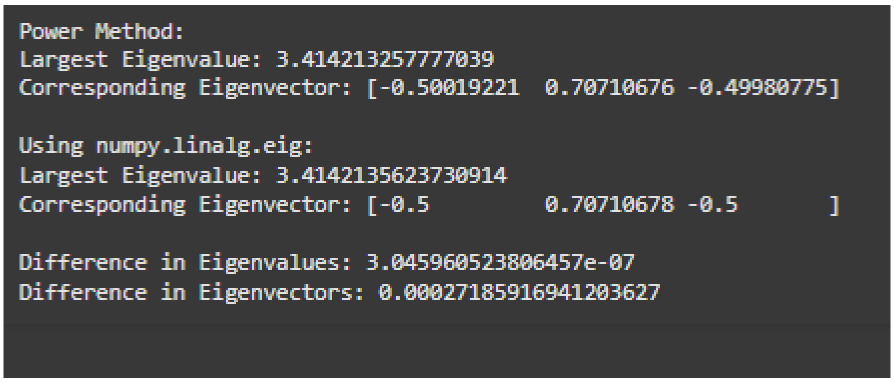


```
Matrix L (Lower Triangular):
 [[ 1.          0.          0.          0.        ]
 [-0.1         1.          0.          0.        ]
 [ 0.2        -0.0733945   1.          0.        ]
 [ 0.          0.27522936 -0.08173077  1.        ]]

Matrix U (Upper Triangular):
 [[10.         -1.          2.          0.        ]
 [ 0.         10.9        -0.8         3.        ]
 [ 0.          0.          9.5412844  -0.77981651]
 [ 0.          0.          0.          7.11057692]]

Solution using LU Factorization:
 [ 0.82758621  1.48275862 -0.89655172  1.20689655]

Solution using numpy.linalg.solve:
 [ 0.82758621  1.48275862 -0.89655172  1.20689655]

Difference between LU Factorization and numpy.linalg.solve: 2.220446049250313e-16
```

Task #3

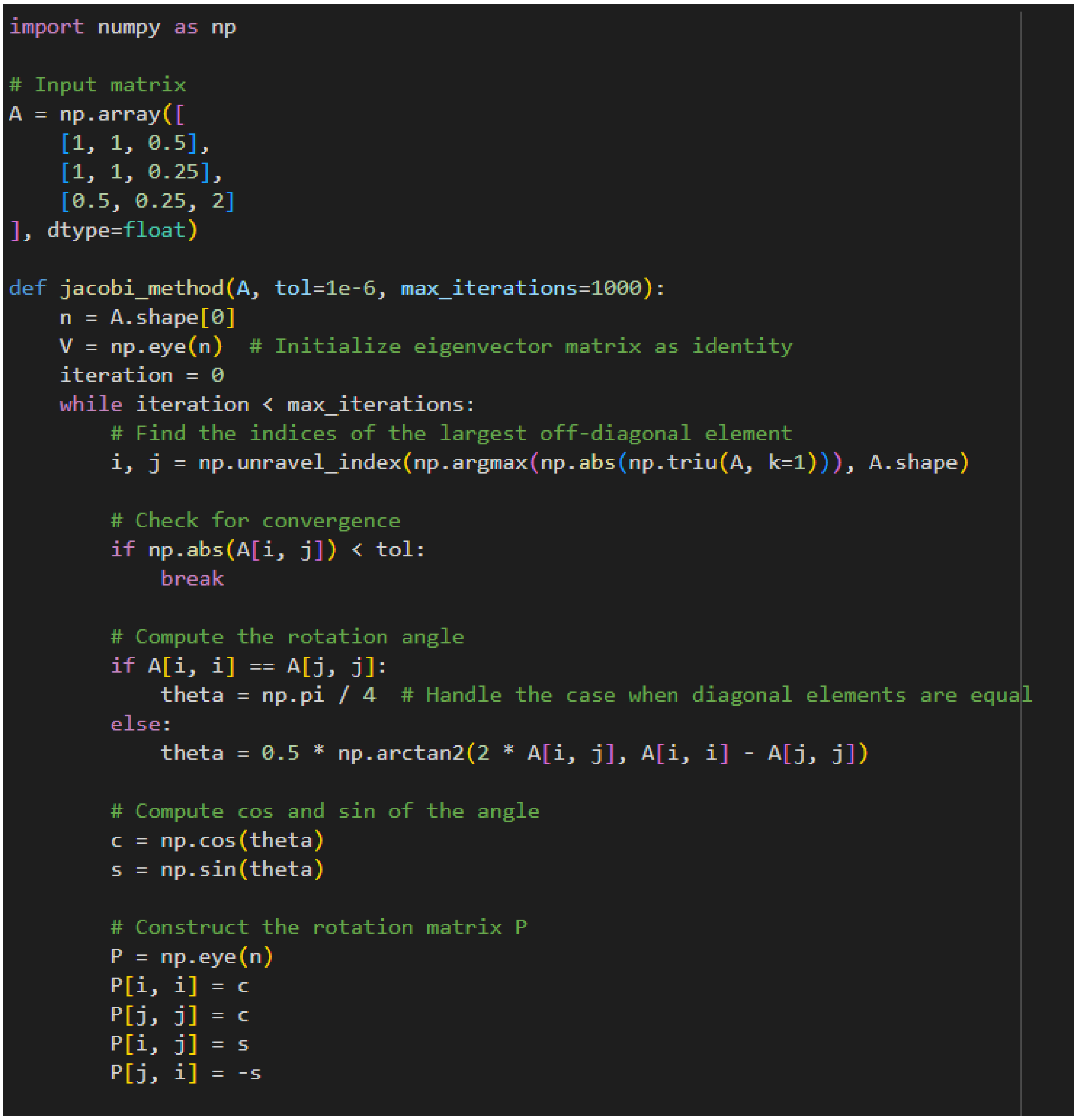


```python
import numpy as np

# Given matrix A
A = np.array([[2, -1, 0],
              [-1, 2, -1],
              [0, -1, 2]], dtype=float)

# Initial vector
v0 = np.array([1, 0, 0], dtype=float)

# Power iteration method
def power_method(A, v0, tol=1e-6, max_iter=1000):
    v = v0 / np.linalg.norm(v0)  # Normalize the initial vector
    lambda_old = 0  # Previous eigenvalue for comparison
    for _ in range(max_iter):
        w = np.dot(A, v)  # Matrix-vector multiplication
        lambda_new = np.dot(v, w)  # Rayleigh quotient
        v_new = w / np.linalg.norm(w)  # Normalize the new vector

        # Check for convergence
        if np.abs(lambda_new - lambda_old) < tol:
            break
        v = v_new
        lambda_old = lambda_new

    return lambda_new, v_new

# Apply power method
lambda_power, v_power = power_method(A, v0)

# Compare with numpy.linalg.eig
eigenvalues, eigenvectors = np.linalg.eig(A)
largest_eigenvalue = np.max(eigenvalues)
largest_eigenvector = eigenvectors[:, np.argmax(eigenvalues)]

# Correct the sign of eigenvectors for comparison
if np.dot(v_power, largest_eigenvector) < 0:
    v_power = -v_power
```

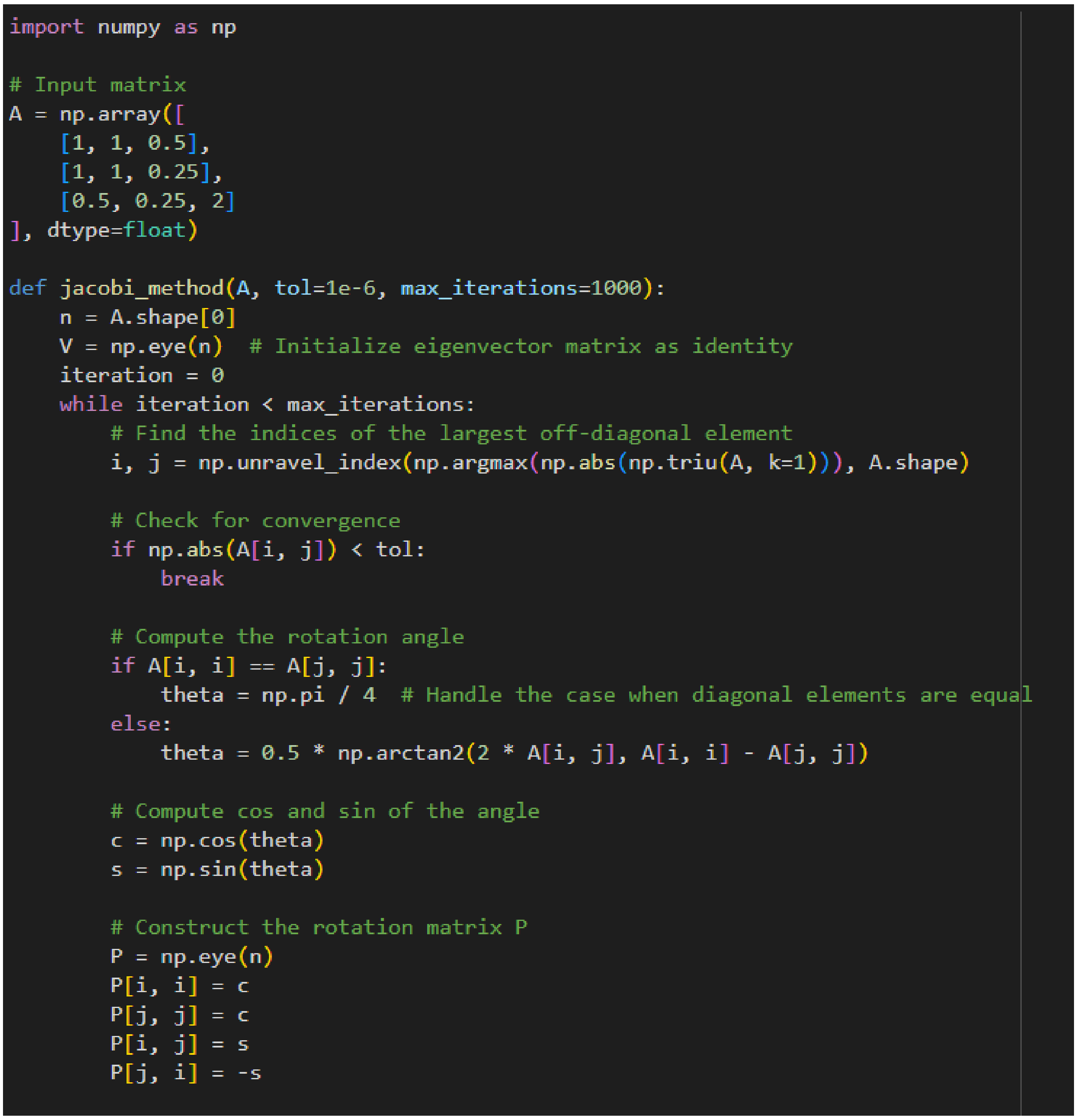


```
Power Method:
Largest Eigenvalue: 3.414213257777039
Corresponding Eigenvector: [-0.50019221  0.70710676 -0.49980775]

Using numpy.linalg.eig:
Largest Eigenvalue: 3.4142135623730914
Corresponding Eigenvector: [-0.5         0.70710678 -0.5       ]

Difference in Eigenvalues: 3.045960523806457e-07
Difference in Eigenvectors: 0.00027185916941203627
```

Task #4



```python
import numpy as np

# Input matrix
A = np.array([
    [1, 1, 0.5],
    [1, 1, 0.25],
    [0.5, 0.25, 2]
], dtype=float)

def jacobi_method(A, tol=1e-6, max_iterations=1000):
    n = A.shape[0]
    V = np.eye(n)  # Initialize eigenvector matrix as identity
    iteration = 0
    while iteration < max_iterations:
        # Find the indices of the largest off-diagonal element
        i, j = np.unravel_index(np.argmax(np.abs(np.triu(A, k=1))), A.shape)

        # Check for convergence
        if np.abs(A[i, j]) < tol:
            break

        # Compute the rotation angle
        if A[i, i] == A[j, j]:
            theta = np.pi / 4  # Handle the case when diagonal elements are equal
        else:
            theta = 0.5 * np.arctan2(2 * A[i, j], A[i, i] - A[j, j])

        # Compute cos and sin of the angle
        c = np.cos(theta)
        s = np.sin(theta)

        # Construct the rotation matrix P
        P = np.eye(n)
        P[i, i] = c
        P[j, j] = c
        P[i, j] = s
        P[j, i] = -s
```



```python
import numpy as np

# Input matrix
A = np.array([
    [1, 1, 0.5],
    [1, 1, 0.25],
    [0.5, 0.25, 2]
], dtype=float)

def jacobi_method(A, tol=1e-6, max_iterations=1000):
    n = A.shape[0]
    V = np.eye(n)  # Initialize eigenvector matrix as identity
    iteration = 0
    while iteration < max_iterations:
        # Find the indices of the largest off-diagonal element
        i, j = np.unravel_index(np.argmax(np.abs(np.triu(A, k=1))), A.shape)

        # Check for convergence
        if np.abs(A[i, j]) < tol:
            break

        # Compute the rotation angle
        if A[i, i] == A[j, j]:
            theta = np.pi / 4  # Handle the case when diagonal elements are equal
        else:
            theta = 0.5 * np.arctan2(2 * A[i, j], A[i, i] - A[j, j])

        # Compute cos and sin of the angle
        c = np.cos(theta)
        s = np.sin(theta)

        # Construct the rotation matrix P
        P = np.eye(n)
        P[i, i] = c
        P[j, j] = c
        P[i, j] = s
        P[j, i] = -s
```